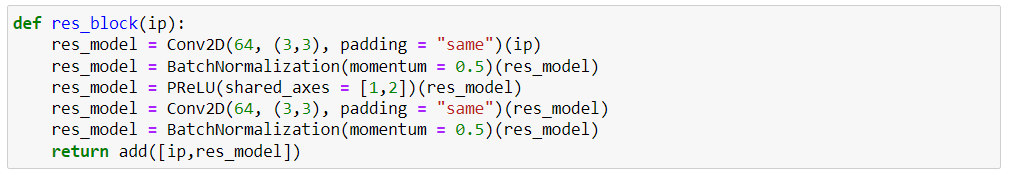


This is a Python script that imports several libraries and sets up a machine learning model using the TensorFlow and Keras libraries.

The os library is used for interacting with the operating system. The cv2 library is used for image processing and computer vision tasks. The numpy library is used for numerical computing with Python. The matplotlib library is used for creating visualizations in Python. %matplotlib inline is a magic command that allows the user to display visualizations in the notebook. The pprint function is imported from the pprint library for pretty printing data structures. The random library is used for generating random numbers and data. The tensorflow library is used for creating and training machine learning models. tf.get\_logger().setLevel(logging.ERROR) is used to set the logging level for TensorFlow to error only. keras is a high-level API for building and training deep learning models in TensorFlow. Sequential is a class for creating a linear stack of layers in a Keras model. Model is a class for creating a more complex, customizable model in Keras. Conv2D is a 2D convolutional layer that can be added to a Keras model. PReLU is a parametric ReLU activation function that can be added to a Keras model. BatchNormalization is a layer that normalizes the inputs to a neural network, which can help with training. Flatten is a layer that flattens a tensor into a 1D array. UpSampling2D is a layer that upsamples an image by repeating the pixels in a certain pattern. LeakyReLU is a leaky ReLU activation function that can be added to a Keras model. Dense is a fully connected layer that can be added to a Keras model. Input is a function for creating an input layer for a Keras model. add is a function for adding layers together in a Keras model. VGG19 is a pre-trained deep learning model for image classification tasks, which can be used as a feature extractor.

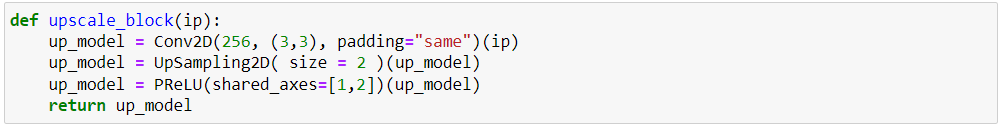


This is a Python function that defines a residual block for a deep learning model that uses convolutional neural networks (CNNs). Residual blocks are a common building block for deep CNNs and help to improve the performance of the model by making it easier to train and reducing the risk of vanishing gradients.

The input argument ip represents the input tensor to the residual block.

The function defines the following operations:

1. res\_model = Conv2D(64, (3,3), padding = "same")(ip): Applies a 2D convolutional layer to the input tensor with a filter size of 3x3, 64 output channels, and padding set to "same" to preserve the input shape. The resulting tensor is stored in res\_model.
2. res\_model = BatchNormalization(momentum = 0.5)(res\_model): Applies batch normalization to res\_model, which normalizes the inputs to the layer and helps with training. The momentum parameter controls how quickly the moving averages of the mean and variance are updated during training.
3. res\_model = PReLU(shared\_axes = [1,2])(res\_model): Applies a parametric ReLU activation function to res\_model, which introduces a learnable parameter that helps to avoid the "dying ReLU" problem. The shared\_axes parameter specifies which axes to share the same parameter over; in this case, it is shared over the second and third axes (i.e., the height and width dimensions).
4. res\_model = Conv2D(64, (3,3), padding = "same")(res\_model): Applies another 2D convolutional layer to res\_model with the same settings as the previous one. The resulting tensor is stored in res\_model.
5. res\_model = BatchNormalization(momentum = 0.5)(res\_model): Applies batch normalization to res\_model.
6. return add([ip,res\_model]): Adds the input tensor ip to res\_model elementwise, and returns the result. This creates the residual connection that gives the residual block its name, and allows the model to learn the difference between the input and the desired output.



This is a Python function that defines an upscale block for a deep learning model that uses convolutional neural networks (CNNs). Upscale blocks are typically used in image super-resolution tasks, where the goal is to increase the resolution of a low-resolution image.

The input argument ip represents the input tensor to the upscale block.

The function defines the following operations:

1. up\_model = Conv2D(256, (3,3), padding="same")(ip): Applies a 2D convolutional layer to the input tensor with a filter size of 3x3, 256 output channels, and padding set to "same" to preserve the input shape. The resulting tensor is stored in up\_model.
2. up\_model = UpSampling2D(size=2)(up\_model): Upsamples the up\_model tensor by a factor of 2 in both the height and width dimensions using bilinear interpolation. This effectively doubles the resolution of the tensor.
3. up\_model = PReLU(shared\_axes=[1,2])(up\_model): Applies a parametric ReLU activation function to up\_model, which introduces a learnable parameter that helps to avoid the "dying ReLU" problem. The shared\_axes parameter specifies which axes to share the same parameter over; in this case, it is shared over the second and third axes (i.e., the height and width dimensions).
4. return up\_model: Returns the resulting tensor up\_model. This tensor has twice the resolution of the input tensor and can be passed to subsequent layers to further increase the resolution of the image.

# Generator:

Super-resolution generator is a deep learning model that aims to increase the resolution of low-resolution images. The generator part of the model consists of multiple layers of convolutional and upsampling layers. The generator takes as input a low-resolution image and produces a high-resolution image as output. The generator is trained in an adversarial manner with a discriminator, which learns to distinguish between real and generated images.

The generator's objective is to produce high-resolution images that are indistinguishable from real high-resolution images, as judged by the discriminator. The generator's loss function is defined as follows:

***L\_G = L2(y\_true, y\_pred) + λ \* L\_adv(y\_pred, y\_true)***

where L2 denotes the mean squared error (MSE) loss, y\_true is the ground truth high-resolution image, y\_pred is the generated high-resolution image, L\_adv is the adversarial loss, and λ is a hyperparameter that controls the relative importance of the MSE and adversarial losses.

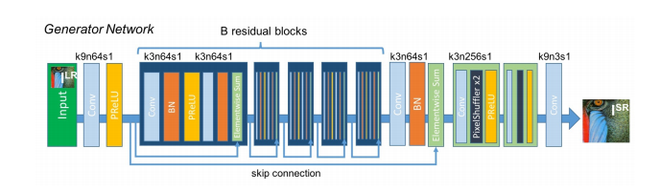
The adversarial loss is defined using the binary cross-entropy loss, and it measures how well the generator can fool the discriminator:

***L\_adv(y\_pred, y\_true) = -log(D(G(x)))***

where G(x) is the generated image, D(.) is the discriminator function, and log is the natural logarithm. The generator tries to minimize this loss by generating images that are classified as real by the discriminator.

The generator is typically implemented as a deep convolutional neural network (CNN) with multiple layers of convolutional and upsampling layers. Each layer applies a convolution operation followed by an activation function, such as the rectified linear unit (ReLU). The last layer of the generator typically uses a tanh activation function to produce an image with pixel values in the range [-1, 1]. Upsampling is often performed using bilinear interpolation or nearest neighbor interpolation. In addition, skip connections can be used to connect earlier layers to later layers to help preserve low-level image features.

The generator is trained using stochastic gradient descent (SGD) or one of its variants, such as Adam. The gradients of the generator's loss with respect to its parameters are computed using backpropagation through the network, and the weights are updated to minimize the loss. The generator is typically trained in conjunction with the discriminator, with the two models being trained iteratively in a min-max game until convergence is reached.





This Python function creates a generator model for super-resolution image generation. The function takes two input arguments: gen\_ip and num\_res\_block.

gen\_ip is the input tensor to the generator model, and num\_res\_block is the number of residual blocks to include in the model.

The function starts by applying a 2D convolutional layer to the input tensor with 64 filters of size 9x9 and padding set to "same". The output of this layer is then passed through a Parametric ReLU activation function (PReLU) with shared axes set to [1,2]. The output of this layer is stored in the temp variable.

Next, a loop is executed num\_res\_block times, with each iteration calling the res\_block function to add a residual block to the model. The res\_block function applies two 2D convolutional layers with 64 filters each and padding set to "same", separated by a Batch Normalization layer with momentum set to 0.5. The output of the second convolutional layer is then added to the input tensor of the block, forming the residual connection.

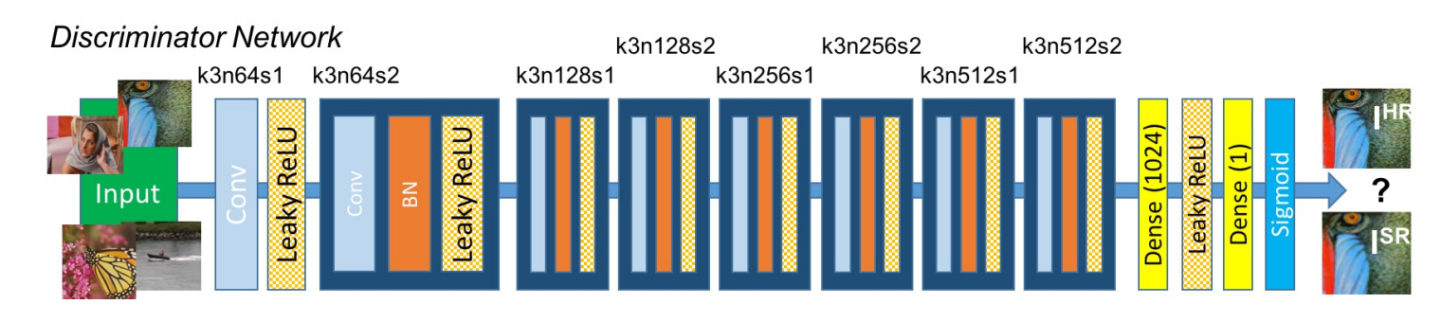
After all the residual blocks have been added to the model, another 2D convolutional layer with 64 filters of size 3x3 and padding set to "same" is applied to the output tensor. The output of this layer is passed through a Batch Normalization layer with momentum set to 0.5, and then added to the temp variable from earlier, forming another residual connection.

Two upscale\_block functions are then called to perform upsampling on the output tensor. The upscale\_block function applies a 2D convolutional layer with 256 filters of size 3x3 and padding set to "same", followed by an upsampling layer that doubles the height and width of the input tensor using nearest neighbor interpolation. The output of the upsampling layer is then passed through a PReLU activation function with shared axes set to [1,2].

Finally, a 2D convolutional layer with 3 filters of size 9x9 and padding set to "same" is applied to the output tensor to produce the final output image.

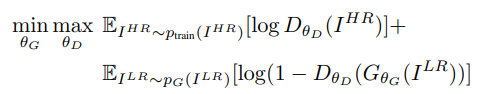
The function returns a Keras Model object that takes the input tensor gen\_ip and produces the output tensor op.

# Discriminator:

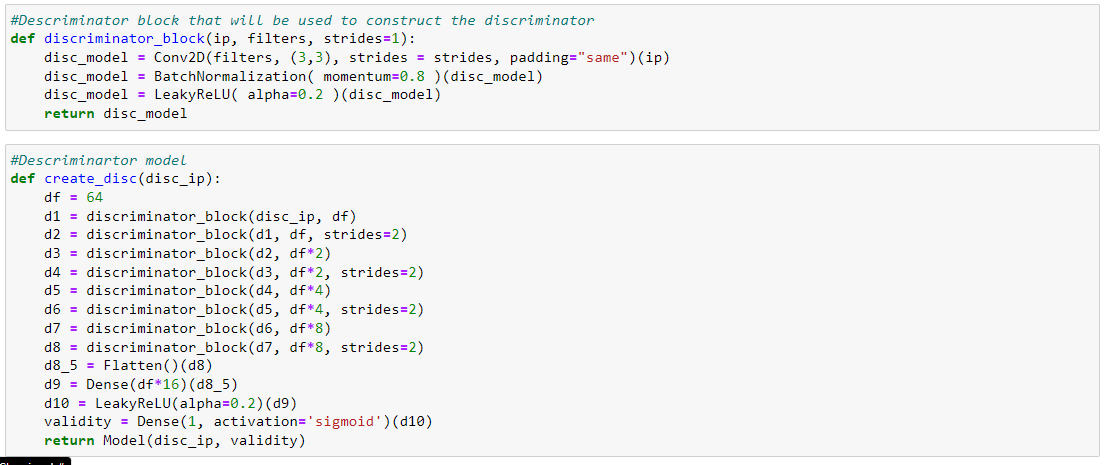


The discriminator architecture is constructed in the best way to support a typical GAN procedure. Both the generator and discriminator are competing with each other, and they are both improving simultaneously. While the discriminator network tries to find the fake images, the generator tries to produce realistic images so that it can escape the detection from the discriminator. The working in the case of SRGANs is similar as well, where the generative model G with the goal of fooling a differentiable discriminator D that is trained to distinguish super-resolved images from real images.

Hence the discriminator architecture shown in the above image works to differentiate between the super-resolution images and the real images. The discriminator model that is constructed aims to solve the adversarial min-max problem. The general idea for the formulation of this equation can be interpreted as follows:



The discriminator architecture constructed is quite intuitive and easy to understand. We make use of an initial convolutional layer followed by a Leaky ReLU activation function. The alpha value for the Leaky ReLU is set to 0.2 for this structure. Then we have a bunch of repeating blocks of convolutional layers, followed by the batch normalization layer and the Leaky ReLU activation function. Once you have five of these repetitive blocks, we have the dense layers followed by the sigmoid activation function for performing the classification action. Note that the initial starting convolutional size is 64 x 64, which is multiplied by 2 after two complete blocks each until we reach the 8x upscaling factor of 512 x 512. This discriminator model helps the generator to learn more effectively and produce better results.



These Python codes define functions for creating a discriminator block and a discriminator model in a deep learning framework like Keras or TensorFlow. The purpose of the discriminator model is to classify images as real or fake in a generative adversarial network (GAN).

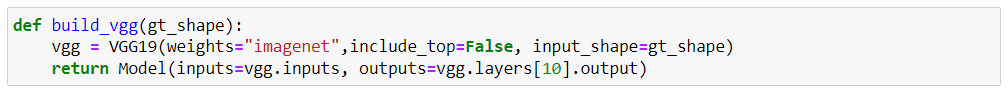
The discriminator\_block function takes three arguments: ip for the input layer, filters for the number of filters in the convolutional layer, and strides for the stride size of the convolution. It creates a convolutional layer with a 3x3 kernel size, applies batch normalization to the output of the convolution, and applies a leaky ReLU activation function with a slope of 0.2 to introduce non-linearity.

The create\_disc function takes one argument disc\_ip which represents the input layer of the discriminator model. It initializes a variable df to 64 and creates the discriminator model by stacking several discriminator blocks defined by the discriminator\_block function.

The discriminator model consists of several layers of convolutional and dense layers. The d1 to d8 variables represent the output of each discriminator block in the model. The output of the last discriminator block, d8, is flattened using the Flatten() function, and fed into a dense layer d9 with a size of df\*16, followed by a leaky ReLU activation function d10. Finally, the output is passed through a dense layer validity with a single output unit and a sigmoid activation function, which produces a probability score for whether the input image is real or fake.

Overall, these functions define a discriminator model that can be trained in conjunction with a generator model in a GAN to generate realistic images. The discriminator model takes in an image and classifies it as real or fake, and the generator model attempts to generate images that are classified as real by the discriminator model. The training process involves adjusting the weights of both models in order to improve the quality of the generated images over time.

# Feature extraction using VGG19:



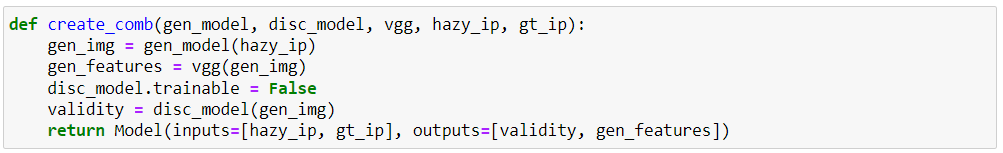
This Python code defines a function called build\_vgg that creates a pre-trained VGG19 convolutional neural network model and returns a modified version of it.

The VGG19 model is a deep neural network that was trained on the ImageNet dataset for image classification. The weights parameter is set to "imagenet" which means that the model will be initialized with pre-trained weights on the ImageNet dataset. The include\_top parameter is set to False which means that the top fully-connected layers of the network that are responsible for image classification will be removed. The input\_shape parameter specifies the shape of the input image that the network will be expecting.

The modified version of the model that is returned by the build\_vgg function includes all of the layers of the pre-trained VGG19 model up to layer 10, but with the final fully-connected layers removed. This modified model can be used as a feature extractor to extract features from an input image, which can then be used for other purposes such as image segmentation, object detection, or image retrieval.

The Model function is used to create a new Keras model object that takes the input and output of the pre-trained VGG19 model as its input and output. The inputs parameter is set to vgg.inputs which represents the input layer of the pre-trained VGG19 model. The outputs parameter is set to vgg.layers[10].output which represents the output of layer 10 of the pre-trained VGG19 model. This modified model can then be used to extract features from input images that can be used for other purposes, such as training a separate model for image classification or object detection.

# Model customization:



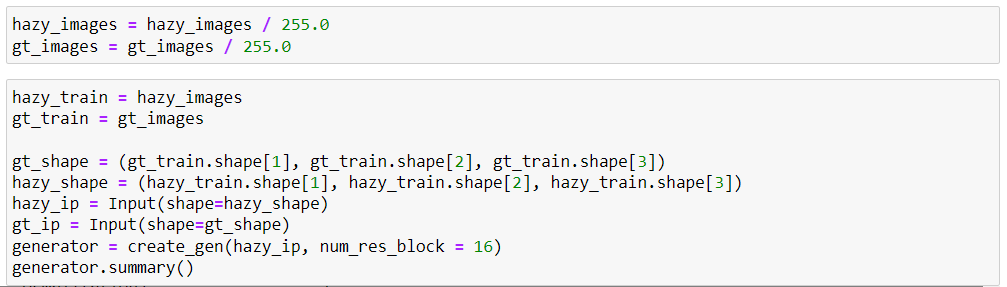
The Python code defines a function called create\_comb that takes several inputs and returns a Keras model object.

The function takes the following inputs:

* gen\_model: A Keras model object that represents a generator network.
* disc\_model: A Keras model object that represents a discriminator network.
* vgg: A Keras model object that represents a VGG19 network.
* hazy\_ip: A Keras input tensor that represents the input image to the generator network.
* gt\_ip: A Keras input tensor that represents the ground truth image.

The function first applies the generator network gen\_model to the hazy\_ip input tensor to generate a new image gen\_img. Then, the vgg model is applied to gen\_img to extract features from the generated image. The disc\_model.trainable is set to False to freeze the weights of the discriminator network during training. Next, the disc\_model is applied to gen\_img to obtain the output validity, which represents the discriminator's prediction of whether gen\_img is a real or fake image.

Finally, the function returns a Keras model object that takes hazy\_ip and gt\_ip as inputs and outputs the validity and gen\_features tensors. The gen\_features tensor represents the features extracted from the generated image using the VGG19 network. This model is used for training a GAN-based image dehazing network, where the generator network is trained to generate clear images from hazy images, and the discriminator network is trained to distinguish between the generated clear images and the ground truth clear images. The VGG19 network is used to calculate the perceptual loss between the generated clear images and the ground truth clear images.



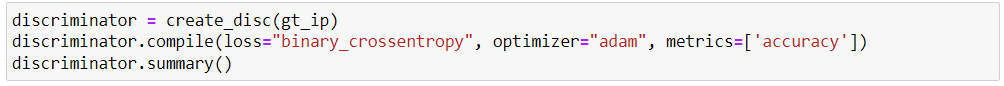
This Python code preprocesses the input images and defines a Keras model object for a generator network. First, the input images hazy\_images and gt\_images are normalized by dividing them by 255.0. This is a common normalization technique used for image data. Next, the normalized images are assigned to hazy\_train and gt\_train variables, respectively. The gt\_shape variable is then defined as a tuple containing the height, width, and number of channels of the gt\_train images.

Similarly, the hazy\_shape variable is defined as a tuple containing the height, width, and number of channels of the hazy\_train images.

An input tensor hazy\_ip is then defined using the hazy\_shape variable, representing the input image to the generator network. Similarly, an input tensor gt\_ip is defined using the gt\_shape variable, representing the ground truth image.

Finally, a generator network is created using the create\_gen function with hazy\_ip as input and the number of residual blocks set to 16. The generator variable represents the resulting Keras model object, which is then summarized using the summary() method.

This code is commonly used for training a deep learning model for image dehazing, where the goal is to generate clear images from hazy images. The generator network is responsible for generating clear images, while the ground truth images are used to calculate the loss during training.



This Python code appears to be related to building and training a deep learning model, specifically a discriminator model. Here is a breakdown of what each line does:

* discriminator = create\_disc(gt\_ip) - This line creates a new discriminator model using the create\_disc function, passing in a parameter gt\_ip as an input. The details of this function are not shown here, but it likely constructs a neural network using a specific architecture and returns the resulting model.
* discriminator.compile(loss="binary\_crossentropy", optimizer="adam", metrics=['accuracy']) - This line compiles the discriminator model, specifying the loss function to be "binary\_crossentropy", the optimizer to be "adam", and the evaluation metric to be the accuracy of the predictions.
* discriminator.summary() - This line prints a summary of the discriminator model to the console. This summary includes the number of layers in the model, the number of parameters in each layer, and the output shape of each layer. This can be useful for debugging and understanding the architecture of the model.

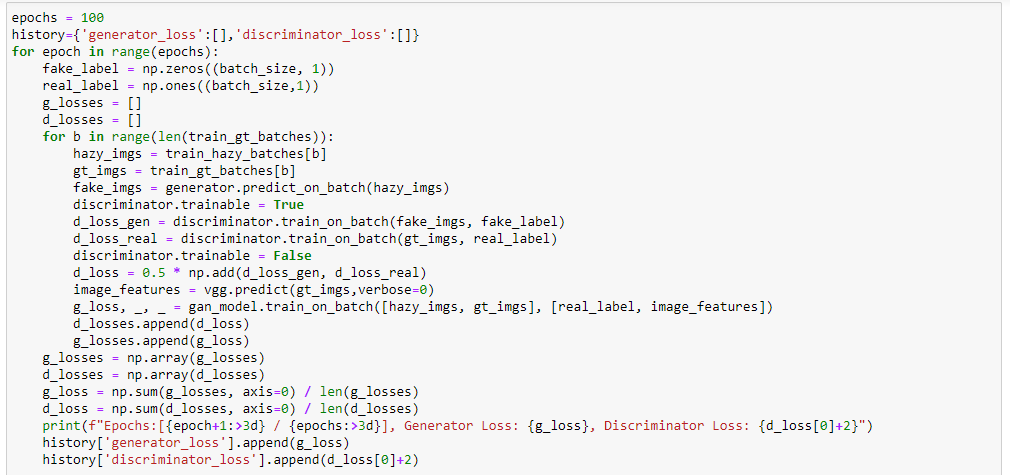
Overall, this code creates and compiles a discriminator model for use in a larger deep learning pipeline, likely for the purpose of performing binary classification on some input data.



This Python code appears to be using a deep learning library, likely TensorFlow or Keras, to build and summarize a pre-trained VGG (Visual Geometry Group) convolutional neural network for image classification.

Here's a breakdown of what each line of code does:

* vgg = build\_vgg((512,512,3)): This line of code builds a VGG neural network model with a 512x512 pixel input image size and 3 color channels. The build\_vgg() function is likely defined elsewhere in the codebase and constructs the VGG architecture, which consists of a series of convolutional layers and max pooling layers followed by several fully connected layers.
* vgg.trainable = False: This line of code sets the trainable attribute of the VGG model to False. This means that when the model is used later on for training, the weights of the VGG layers will not be updated during the training process. This is often done when using pre-trained models as a starting point for transfer learning, where the weights of the pre-trained model are frozen to avoid overfitting and to speed up training.
* print(vgg.summary()): This line of code prints a summary of the VGG model's architecture to the console. The summary includes information about the shape and number of parameters of each layer in the model, as well as the total number of trainable parameters in the model. This information can be useful for debugging and understanding the structure of the model.



The for loop runs for the number of epochs specified. For each epoch, the generator and discriminator losses for that epoch are stored in the `history` dictionary. Inside the epoch loop, the code initializes `fake\_label` and `real\_label` numpy arrays of shape `(batch\_size, 1)` with all zeros and ones respectively. These labels are used to train the discriminator and generator. The next lines initialize empty lists for storing the generator and discriminator losses for each batch of data. The inner loop runs for each batch of training data. For each batch, the code loads the hazy and ground truth images for that batch, and generates fake images using the generator. The discriminator is then trained on the fake and real images, and the losses are computed and stored. Next, the generator is trained using the computed losses and the ground truth images. The generator loss and discriminator loss for the current batch are appended to their respective lists. At the end of the inner loop, the generator and discriminator losses are averaged over all batches and printed for the current epoch. The generator and discriminator losses for the epoch are stored in the `history` dictionary. After printing the losses for the current epoch, the code generates and displays a random set of hazy, predicted, and ground truth images using the `plt` module. This provides a visual indication of how well the model is performing. Finally, the code saves the generator model every 50 epochs. The `train\_hazy\_batches`, `train\_gt\_batches`, `generator`, `discriminator`, and `vgg` are already defined and initialized before running this code.